**Functional Programming and Lambda**

Functional programming is a paradigm change.

A functional interface is an interface with only one abstract method all other method will be either default or static. We can annotate the interface with @FunctionalInterface, but it’s a marker annotation only.

That single method is called function descriptor.

Rather creating Anonymous class implementations, we can use lambda expression if the interface is a functional interface.

We can also create our own functional interface, but for simplicity java developers had already created many functional interfaces.

Some of the useful functional interfaces are

Runnable

Predicate<T>

BiPredicate<T,R>

Function<T,R>

BiFunction<T,R,S>

Consumer<T>

BiConsumer<T,R>

Supplier<T>

UnaryOperator<T>

BinaryOperator<T>

There are some other functional interfaces to work with the primitive types like IntBinaryOperator, IntConsumer, IntFunction, IntPredicate, IntSupplier, IntToDoubleFunction, IntToLongFunction. As the upper functional interfaces works with Object types mainly, so to use stream of Integers there will be autoboxing and unboxing which is in efficient, that’s why these primitive functional interfaces are also used.

We can directly instantiate a functional interface with lambda expression like the following

**Predicate<Integer> predicate = num -> num % 2 == 0;**

**Consumer<Integer> consumer = System.out::println;**

**num -> num % 2 == 0** is a lambda expression, with which we can direct add the code here.

If there is only one parameter, then we don’t need parentheses. If there one statement, then we don’t need to add it in curly braces. We don’t need to specify the type of the input it will be inferred automatically.

**System.out::println** is a method reference.

We can use method reference for static methods (like **Math::max** ), member methods (like **String::toUpperCase** ) or constructor (like **String::new** ) . Just that the signature should match.

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

numbers

.stream()

.filter(num -> num % 2 == 0)

.map(num -> num \* 2)

.forEach(System.***out***::println);

.stream() method makes a iterable to stream object

.filter() method takes predicate functional interface to filter out the object. It will take an object and return Boolean.

.map() method takes Function functional interface to transform the object to another object. It will take an object and return another or same object.

.forEach() method takes Consumer functional interface to consume an object. It is void return type. It will only take an object. It is a terminal operation.

.reduce is a reduction operation which takes an identity or a stating element and a BiFunction functional interface which takes two inuts and returns one output. We can use multiple cores and thread to perform reduce if it’s a big set of data.

Integer sum = numbers

.stream()

.reduce(0, (a, b) -> a + b);

0, (a, b) here 0 is the identity or the starting point, a is the aggregate or the output of the previous operation and b is the current item. If we are only doing sum then we can use Integer:: sum or the sum() method.

.distinct() method removes all the duplicate elements.

.sorted() method makes the stream sorted. By default, it will sort in increasing manner. This is an overloaded method which takes Comparator functional interface. We can also assign our custom sorting logic with this. We can also assign Comparator.naturalOrder() or Comparator.reverseOrder().

We can also assign our own logic using Comparator.comparing like sorted

(Comparator.comparing( str -> str.length() ) )

numbers

.stream()

.distinct()

.sorted(Comparator.*naturalOrder*())

//.sorted(Comparator.reverseOrder())

//sorted()

.forEach(System.***out***::println);

.collect(collection) is a terminal operation where we can collect the stream of data and make it as list or set or map anything.

List<Double> squareRoots =numbers

.stream()

.map(Math::sqrt)

.collect(Collectors.toList());

Intermediate operation returns a Stream. Terminal operation returns anything other than Stream.

.allMatch(predicate) ->

.noneMatch(predicate) ->

.anyMatch(predicate) ->

System.out.println(

courses

.stream()

.sorted(Comparator.comparing(Course::getNoOfStudents)

.thenComparing(Course::getName))

.collect(Collectors.toList())

);

Here the sorted function takes a comparator. The comparator first compares with the number of students then among them it compares with the name. We can achieve this with Comparator.comparing method and then thenComparing method.

If we want the reverse then we can use reversed() method.

Comparator<Course> compartor = Comparator.*comparing*(Course::getNoOfStudents)

.reversed()

.thenComparing(Course::getName);

We have comparingInt comparingLong etc and similarly we have thenComparingInt thenComparingLong methods which can improve the performance by by eliminating the usage of autoboxing and unboxing.

There are 2 other methods that we can use along with sorted method that is limit and skip. As name suggested limit method limits the resultant list size and skip method skip the first n records.

With sort, skip and limit we can implement the pagination.

.takeWhile(predicate) -> as the name suggests, it will take elements from the stream while the condition is met and after that it will not take anything. If one element breaks the criteria it will stop there.

.dropWhile(predicate) -> it is just opposite of takeWhile method. It will drop elements until there is an element which does not match the predicate.

System.***out***.println(

courses

.stream()

.sorted(Comparator.*comparingInt*(Course::getReviewScore))

.dropWhile(course -> Course.*isReviewScoreGreaterThan*(course,95))

.collect(Collectors.*toList*())

);

System.***out***.println(

courses

.stream()

.sorted(Comparator.*comparingInt*(Course::getReviewScore))

.takeWhile(course -> Course.*isReviewScoreGreaterThan*(course,95))

.collect(Collectors.*toList*())

);

System.***out***.println(

courses

.stream()

.sorted(Comparator.*comparingInt*(Course::getNoOfStudents))

.limit(1)

.skip(2)

.collect(Collectors.*toList*())

);

System.***out***.println(

courses

.stream()

.sorted(Comparator.*comparingInt*(Course::getNoOfStudents))

.skip(2)

.collect(Collectors.*toList*())

);

System.***out***.println(

courses

.stream()

.sorted(Comparator.*comparingInt*(Course::getNoOfStudents))

.limit(2)

.collect(Collectors.*toList*())

);

There are some other operations like max(predicate), min(predicate). But it will return an optional object. To get value from optional we need to call with get() method.

Optional is an object typically to handle null. If the optional is empty then we can use orElseGet method or orElseThrow method.

There are some other methods like findFirst() which will return the first element that matches the certain criteria. findAny() is also similar like findFirst(), it will return any object that matches the criteria

There are some other methods like sum and average, max, min which takes on a single property.

System.***out***.println(

courses

.stream()

.mapToInt(Course::getNoOfStudents)

.sum()

);

System.***out***.println(

courses

.stream()

.max(Comparator.*comparingInt*(Course::getReviewScore))

.orElse(**new** Course("Kubernetes","Cloud",95,21000))

);

We have a special type of collector which can group by from a list of map or objects. Collect(Collectors.groupBy(Course::getCatagory)) It will group the list of courses depending on the category.

System.***out***.println(

courses

.stream()

.collect(Collectors.*groupingBy*(Course::getCategory))

);

It will group by the courses based on category find the count for all categories.

System.***out***.println(

courses

.stream() .collect(Collectors.*groupingBy*(Course::getCategory,Collectors.*counting*()))

);

It will group by the courses based on category find the course with max course review score.

System.***out***.println(

courses

.stream()

.collect(Collectors.*groupingBy*( Course::getCategory, Collectors.*maxBy*(Comparator.*comparing*(Course::getReviewScore))

))

);

It will group by the courses based on category then map the list of courses to lit of names.

System.***out***.println(

courses

.stream()

.collect(Collectors.*groupingBy*(

Course::getCategory, Collectors.*mapping*(Course::getName,Collectors.*toList*())

))

);

We can directly create the streams with of() method. We can also create streams of primitive. When we create streams with numbers it will create a stream of Integer but when we create streams with int array it will create streams of IntPipeline

**Stream.*of*(1,2,3,4,5,6); // java.util.stream.ReferencePipeline$Head@29ee9faa**

**Arrays.*stream*(new int[]{1,2,3,4,5,6});//java.util.stream.IntPipeline$Head@14acaea5**

We can use the same functions.

We can create primitive value of streams dynamically.

IntStream.*range*(1, 10).sum();

IntStream.*rangeClosed*(1, 10).sum();

We can create a infinitely long stream but then we have to add limit to stop the stream.

IntStream.iterate(0, e->e+1).limit(10).sum();

We can also use peek() method that will not change the items. It will take a consumer functional interface. We can log the numbers using peek() method.

IntStream.*iterate*(0, e->e+1).limit(10).peek(System.***out***::println).sum();

We can not directly use the collect() method to list as it is a stream of primitive. So, collect it to list we have to use boxed method other it will throw an exception

IntStream.*iterate*(0, e->e+1).limit(10).boxed().collect(Collectors.*toList*());

When we are doing with big numbers calculation we should not use IntStream or LongStream as it will overflow.We can use BigInteger object there.

IntStream.*rangeClosed*(1, 10)

.mapToObj(BigInteger::*valueOf*)

.reduce(BigInteger.***ONE***,BigInteger::multiply);

When